Make Code Parallel guided by Patterns

Using Parallelware Trainer

Manuel Arenaz
manuel.arenaz@appentra.com

PARALLELWARE TRAINER

appentra
make code parallel

Expected workshop learning outcomes

- Learn how to decompose real codes into parallel patterns
  - Have experience decomposing the hydrodynamics code LULESH (from the CORAL benchmark suite) into parallel patterns
- Learn how to parallelize real codes using OpenACC
  - Have experience parallelizing the hydrodynamics code LULESH
  - Have a practical step-by-step approach based on patterns for parallelizing any code
- Learn best practices for parallel programming using OpenACC

Why use patterns to parallelize code?

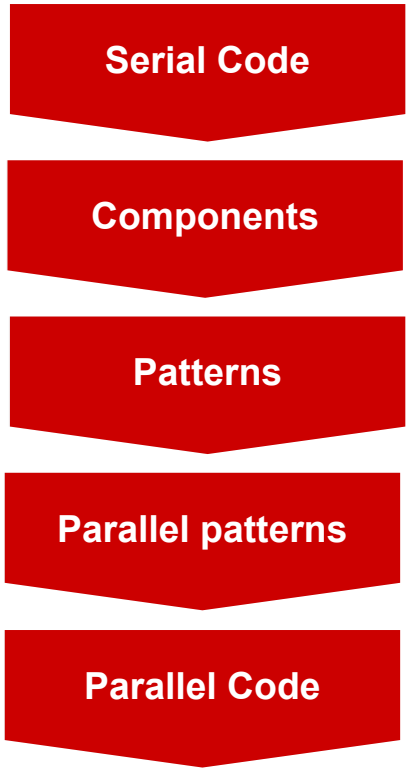
- The OpenACC Application Programming Interface. Version 2.7 (November 2018)
  - "does **not describe automatic detection of parallel regions or automatic offloading of regions of code to an accelerator by a compiler or other tool**."
  - "if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, **the hardware may not guarantee the same result** for each execution."
  - "it is (...) **possible to write a compute region that produces inconsistent numerical results**."
  - "**Programmers need to be very careful that the program uses appropriate synchronization** to ensure that an assignment or modification by a thread on any device to data in shared memory is complete and available before that data is used by another thread on the same or another device."
- **Programmers are responsible for making good use of OpenACC**
- **Decomposition of codes into patterns**
  - Helps to make good use of OpenACC and OpenMP
  - Speeds up the parallelization process
  - Is more likely to result in good performance

Decomposing your code into components

Real codes are large and complex: identifying smaller sections that can be parallelized makes the parallelization task simpler

Types of components:

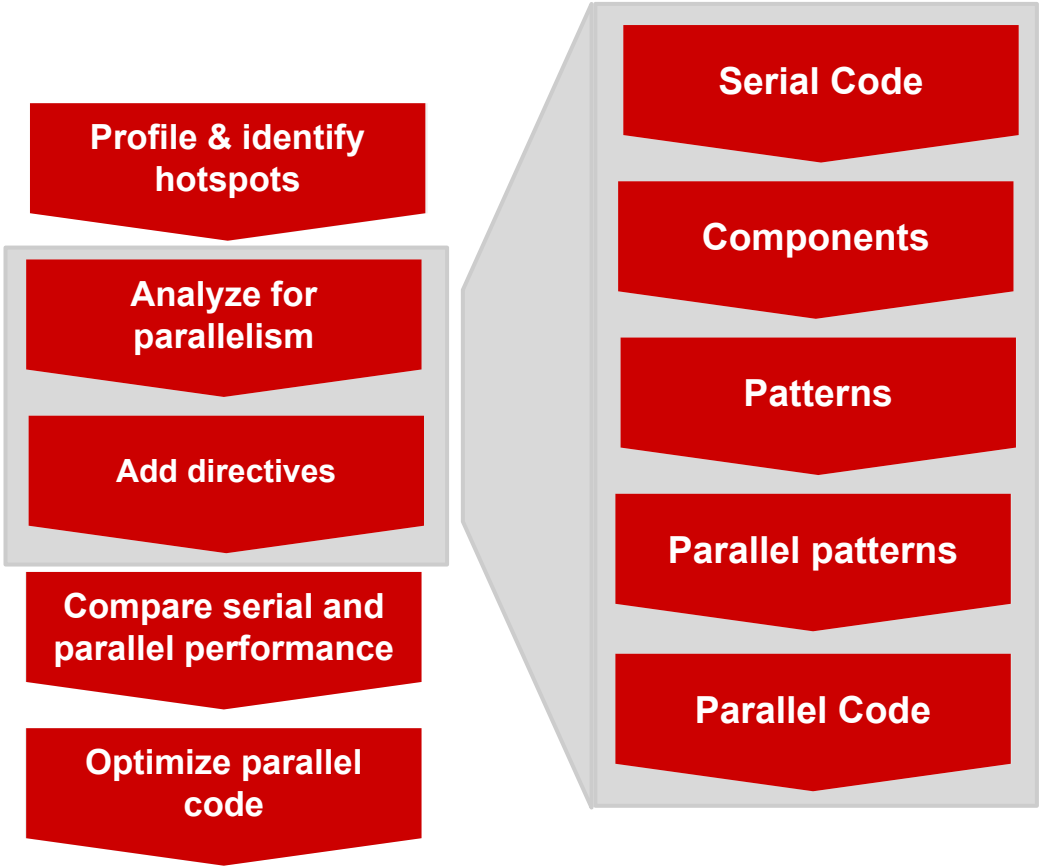
- Scientific components
  (e.g. MATMUL, FFT)
- Code components or patterns
  (e.g. REDUCTION)



Decomposing your code into components

How does it fit into the classical parallelization workflow?

High-productivity approach independent of OpenMP, OpenACC, OmpSs,...



Decomposing your code into components

Step 1: **Use your profiling to**

- Identify calls, routines, functions or loops that consume most of the runtime

Step 2: **For each routine contained in an external library**

- Scientific components: kernels available as external libraries, including but not limited to dense/sparse linear algebra and spectral methods.
- Consider using a highly optimized version of the routine available in the target platform

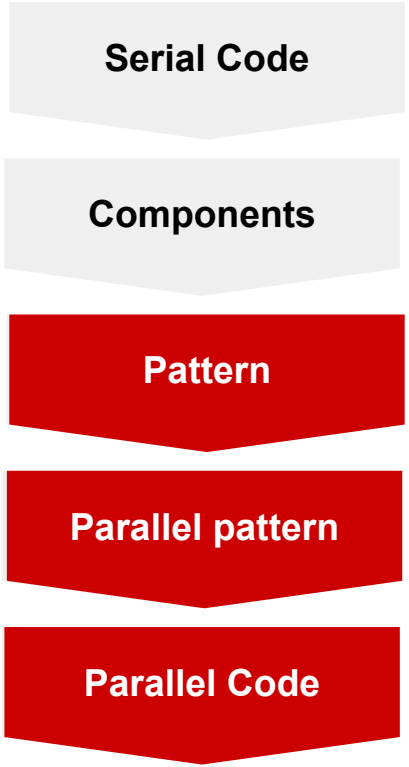
Step 3: **For each routine coded by the programmer that matches a routine contained in external library**

- Consider replacing the corresponding routines with highly-optimized version in your platform

Step 4: **For the remaining user-defined routines**

- Understand the compute patterns and flow patterns you have in your code

Parallelizing by pattern



Identification of parallel patterns

Parallel Patterns

parallel forall

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

parallel scalar reduction

```
for (j=0; j<n; j++ ) {
   A += B[j];
}
```

parallel sparse reduction

```
for (j=0; j<n; j++ ) {
   A[C[j]] += B[j];
}
```

parallel sparse forall

```
for (j=0; j<n; j++ ) {
   A[C[j]] = B[j];
}
```

Forall

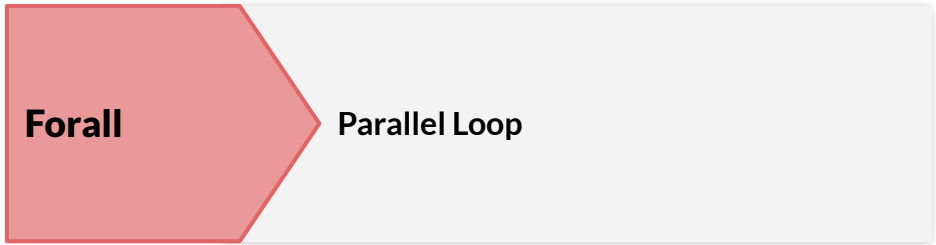
Understanding the sequential code

- A loop that updates the elements of an array.
- Each iteration updates a different element of the array.
- The result of computing this pattern is an array that is the "*output variable*".

parallel forall

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

Identifying opportunities for parallelization



Scalar reduction

Understanding the sequential code

- Combine multiple values into one single element (the scalar reduction variable) by applying an associative, commutative operator.
- Most frequently in a loop
- The result of computing this pattern is a scalar that is the “reduction variable”.

parallel scalar reduction

```
for (j=0; j<n; j++ ) {
   A += B[j];
}
```

Identifying opportunities for parallelization

Scalar reduction

Parallel Loop w/ Built-in reduction
Parallel Loop w/ Atomic
Parallel Loop w/ Explicit Privatization

Sparse reduction

Understanding the sequential code

- A sparse or irregular reduction combines a set of values from a subset of the elements of a vector or array with an associative, commutative operator.
- The set of array elements used cannot be determined until runtime due to the use of subscript array to provide these values.
- The result of computing this pattern is an array that is the “reduction variable”.

parallel sparse reduction

```
for (j=0; j<n; j++ ) {
   A[C[j]] += B[j];
}
```

Identifying opportunities for parallelization

Sparse reduction

Parallel Loop w/ Built-in reduction
Parallel Loop w/ Atomic
Parallel Loop w/ Explicit Privatization

Sparse forall

Understanding the sequential code

- A loop that updates the elements of an array.
- The set of array elements used cannot be determined until runtime due to the use of subscript array to provide these values.
- The result of computing this pattern is an array that is the "*output variable*".

parallel sparse forall

```
for (j=0; j<n; j++ ) {
   A[C[j]] = B[j];
}
```

Identifying opportunities for parallelization

Sparse forall → **Parallel Loop w/ Explicit Privatization**

Why use patterns to parallelize code?

1: Patterns enable to ensure correct variable management in the parallel code

- Each pattern has one output variable that is computed in the code.
- The pattern dictates the correct data scoping of the output variable (e.g. shared in forall, reduction in scalar reduction).

2: Patterns provide algorithmic rules to re-code sequential code into a parallel-equivalent code

- Patterns provide information about the type of computations that are associated with a variable of the code. And this type of computations dictates what codes can be parallelized.
- Examples: pattern scalar reduction can be parallelized, and convergence loop cannot be parallelized

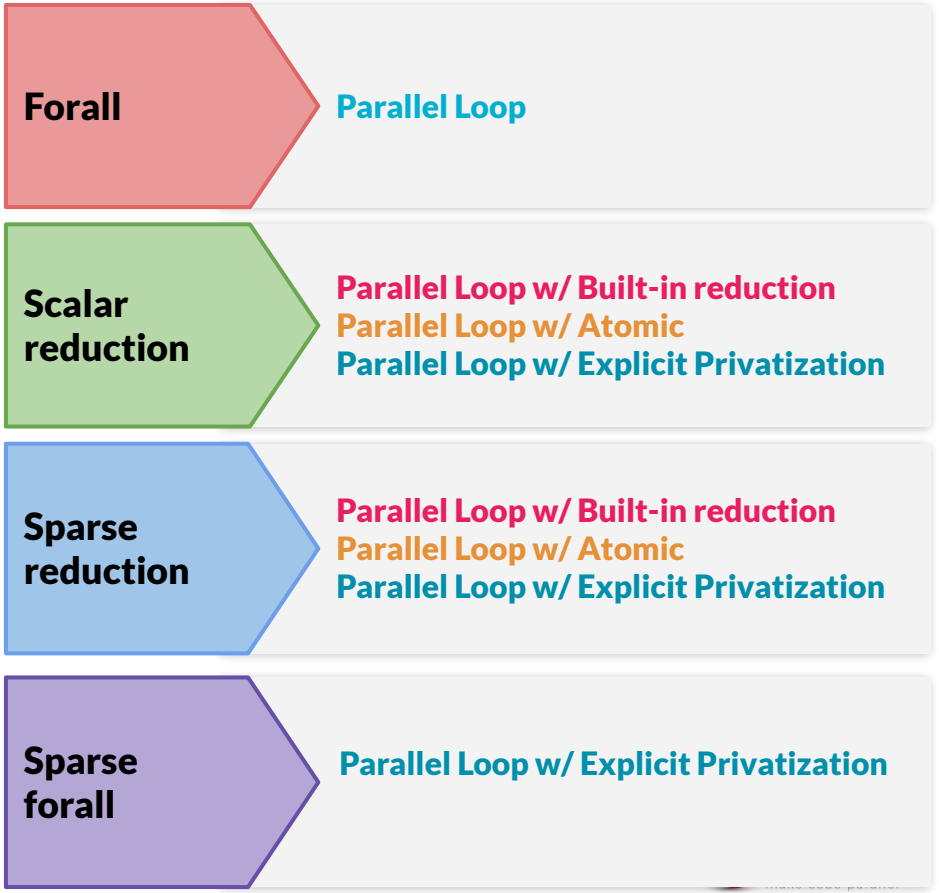
3: Patterns enable to code parallel versions for several standards and platforms

- Each pattern provides code rewriting rules for OpenMP/OpenACC and CPU/GPU.

Parallelization strategies

Patterns and parallelization strategies

Parallelization Strategies



Mapping parallelization strategies to patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

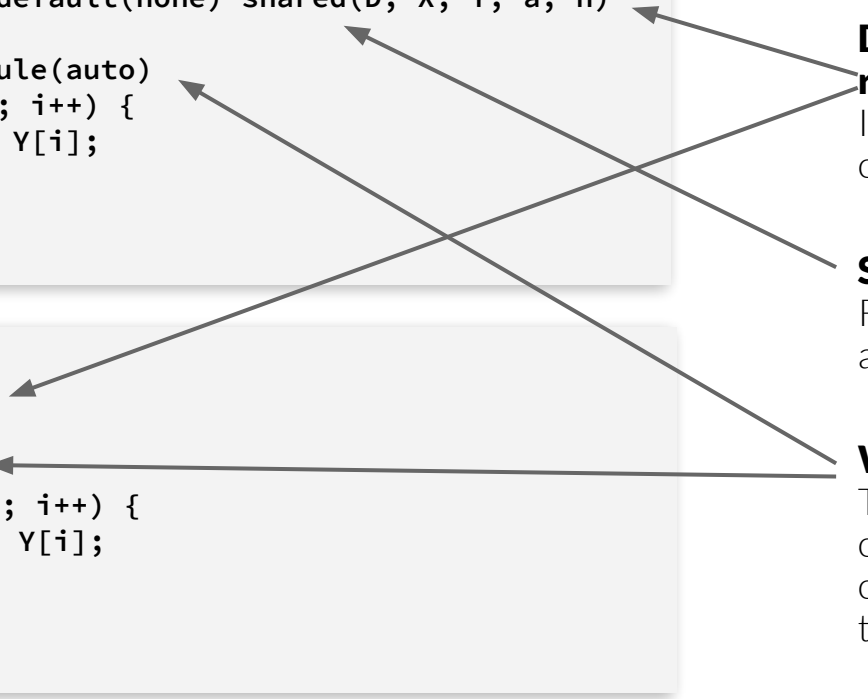
Mapping parallelization strategies to patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

Implementation of Parallel Loop

```
#pragma omp parallel default(none) shared(D, X, Y, a, n)
{
#pragma omp for schedule(auto)
for (int i = 0; i < n; i++) {
    D[i] = a * X[i] + Y[i];
}
} // end parallel
```

```
#pragma acc parallel
{
#pragma acc loop
for (int i = 0; i < n; i++) {
    D[i] = a * X[i] + Y[i];
}
} // end parallel
```



Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop directive allows the compiler to map the computational workload to threads.

Mapping parallelization strategies to patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

Implementation of Parallel Loop w/ Built-in Reduction

```
double sum = 0.0;

#pragma omp parallel default(none) shared(N, sum)
{
#pragma omp for reduction(+: sum) schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum += sqrt(1 - x * x);
}
} // end parallel
```

```
double sum = 0.0;

#pragma acc parallel
{
#pragma acc loop reduction(+: sum)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum += sqrt(1 - x * x);
}
} // end parallel
```

Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop/for directive allows the compiler to map the computational workload to threads.

Reduction
Identifies the loop as a reduction, and identifies the subject of the reduction (i.e. sum) and the reduction operator (i.e. '+')

Mapping parallelization strategies to patterns

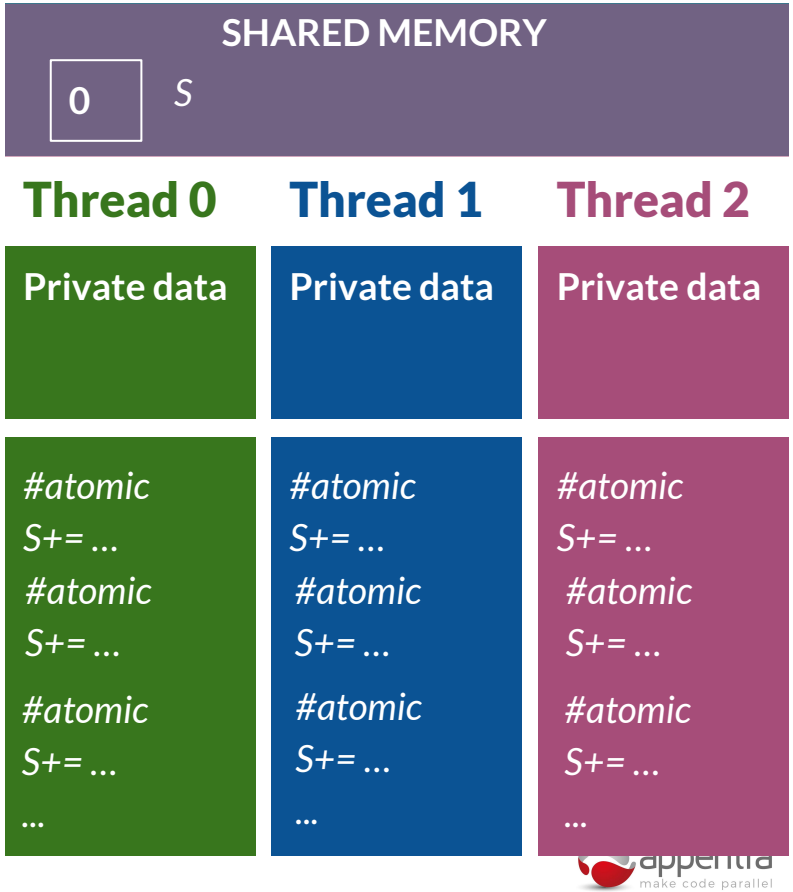
		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

Implementation of Parallel Loop w/ Atomic

Shared variable, *S*, is the 'reduction' variable. No private data.

Access to the variable *S*, is controlled by the 'atomic' directive: i.e. only one thread can read/write the variable at any one time.

In each atomic access of *S*, the thread adds part of the contribution to the total reduction value. In this instance, the reduction operation is an addition.



Implementation of Parallel Loop w/ Atomic

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
{
#pragma omp for schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    #pragma omp atomic update
    sum += sqrt(1 - x * x);
}
} // end parallel
```

```
double sum = 0.0;

#pragma acc parallel
{
#pragma acc loop
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    #pragma acc atomic update
    sum += sqrt(1 - x * x);
}
} // end parallel
```

Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop directive allows the compiler to map the computational workload to threads.

Atomic update
Only one thread can read/write the variable at any one time.

Mapping parallelization strategies to patterns

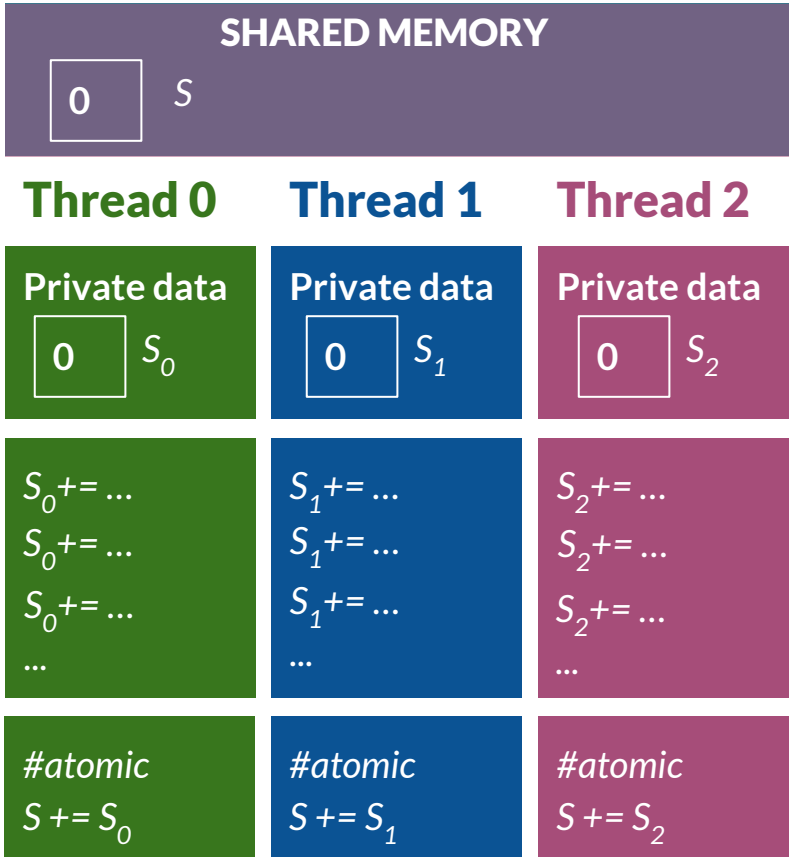
		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

Implementation of Parallel Loop w/ Explicit Privatization

Create private copies $S_0 \ldots S_{p-1}$ of the shared variable S. Initialize the private variables to 0.

Each thread computes a partial sum using its private copy only. No synchronization with other threads.

Each thread adds its partial sum to the global sum. Using *atomic* guarantees exclusive access to the reduction variable.



Implementation of Parallel Loop w/ Explicit Privatization

Create private, local copies
Create thread-local copies of the reduction variable and initialize the local copies to 0.

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
{
// preamble
double sum_private = 0;
// end preamble
#pragma omp for schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum_private += sqrt(1 - x * x);
}
// postamble
#pragma omp atomic update
sum += sum_private;
// end postamble
} // end parallel
```

Explicit privatization
Each thread performs a thread-local computation on the private copy.

Use atomic to contribute to global value
To complete the calculation each thread adds its contribution to the global shared using *atomic*.

```
#pragma omp parallel default(none) shared(col_ind, n, row_ptr, val, x, y)
{
// preamble
unsigned int y_length = 0 + n;
double *y_private = (double *) malloc(sizeof(double) * y_length);
for (int i = 0; i < y_length; ++i) {
  y_private[i] = 0;
}
// end preamble
#pragma omp for schedule(auto)
for (int i = 0; i < n; i++) {
    for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
        y_private[col_ind[k]] = y_private[col_ind[k]] + x[i] * val[k];
    }
}
// postamble
#pragma omp critical
for(int i = 0; i < y_length; ++i) {
  y[i] += y_private[i];
}
free(y_private);
// end postamble
} // end parallel
```

Parallelization strategies
Pros & Cons

Mapping parallelization strategies to patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

	Pros	Cons
Parallel Loop	- Easy to implement - No synchronization overhead within the loop	- Limited applicability: only works when each loop iteration is entirely independent
Parallel Loop w/ Built-in reduction	- Scales with threads/core counts, not the problem size - Offers speedup even for codes with low arithmetic intensity - Complexity handled by the compiler - Potential for highly optimized implementation (compiler/platform dependent)	- Can only be used for supported reduction operators
Parallel Loop w/ atomic protection	- Easy to understand - Provides speedup for codes with high arithmetic intensity - Solution for reduction patterns where operator is not supported by build-in reduction clause	- Synchronization overhead scales with the number of threads - Poor performance for codes with low arithmetic intensity
Parallel Loop w/ explicit privatization	- Possible to achieve speedup similar to Built-in Reductions - Programmer has full control of the parallel implementation	- Significant programmer effort - Not suitable for GPUs due to memory requirements

Tasking
(New in Parallelware Trainer)

Mapping strategies to patterns for Tasking

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Fine-grain tasking on CPU (OpenMP 3.5 task/taskwait; OpenMP 4.5 taskloop -implementation dependent-)					
Parallel Pattern	Forall	✓			
	Scalar Reduction		upcoming	✓	upcoming
	Sparse Reduction			✓	upcoming
	Sparse forall				upcoming
Coarse-grain tasking on CPU (OpenMP 3.5: task/taskwait + loop stripmining; OpenMP 4.5 taskloop grainsize/numtasks)					
Parallel Pattern	Forall	upcoming			
	Scalar Reduction		upcoming	upcoming	upcoming
	Sparse Reduction			upcoming	upcoming
	Sparse forall				

Implementation of Parallel Loop w/ Atomic

OpenMP 3.5: task/taskwait

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp master
{
for (int i = 0; i < N; i++) {
#pragma omp task shared(sum)
   {
    double x = (i + 0.5) / N;
    #pragma omp atomic update
    sum += sqrt(1 - x * x);
   }
}
#pragma omp taskwait
} // end parallel master
```

OpenMP 4.5: taskloop

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp single
{
    #pragma omp taskloop
    for (int i = 0; i < N; i++) {
        double x = (i + 0.5) / N;
        #pragma omp atomic update
        sum += sqrt(1 - x * x);
    }
} // end parallel
```

Implementation of Parallel Loop w/ Built-in Reduction

(upcoming -not available yet-)

With private reduction (OpenMP 5.0)

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N) reduction(+:sum)
#pragma omp master
{
#pragma omp taskloop grainsize(BS) in_reduction(sum)
      for (int i = 0 ; i < N; i++) {
      double x = (i + 0.5) / N;
      sum += sqrt(1 - x * x);
      }
} // end parallel
```

Implementation of Coarse-Grain Tasking

(upcoming -not available yet-)

OpenMP 3.0: task/taskwait + loop stripmining

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp master
{
for (int I = 0; I < N; I+=BS) {
#pragma omp task shared(sum)
   for (int i = I ; i < I+BS && i < N; i++) {
    double x = (i + 0.5) / N;
    #pragma omp atomic update
    sum += sqrt(1 - x * x);
   }
}
#pragma omp taskwait
} // end parallel master
```

Implementation of Coarse-Grain Tasking Parallel Loop w/ Explicit Privatization

(upcoming -not available yet-)

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp master
{
for (int I = 0; I < N; I+=BS) {
#pragma omp task shared(sum)
   {
    double psum = 0.0;
    for (int i = I ; i < I+BS && i < N; i++) {
     double x = (i + 0.5) / N;
     psum += sqrt(1 - x * x);
    }
    #pragma omp atomic update
    sum += psum;
   }
}
#pragma omp taskwait
} // end parallel
```

Use cases

Case studies: decomposing PI and LULESH microkernel

GOALS:

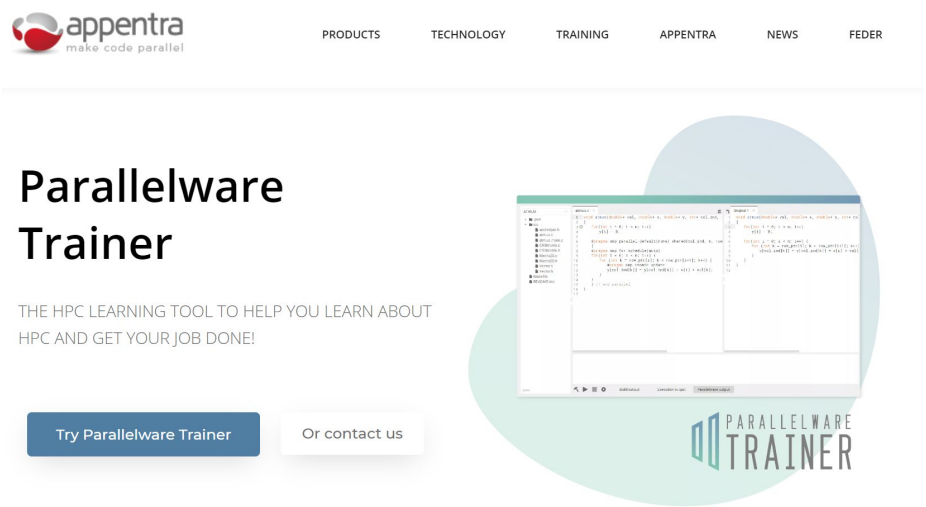
- Understand how to split PI up into components
- Understand how to split LULESH into components

HANDS-ON LAB

- **Install and launch Parallelware Trainer:**
  https://www.appentra.com/
- **Two exercises:**
  - Simple walkthrough: parallelizing the calculation of π
  - Parallelizing a micro-kernel of the CORAL-lulesh benchmark

➢ **Follow the instructions on the worksheet**

PARALLELWARE TRAINER



appentra
make code parallel

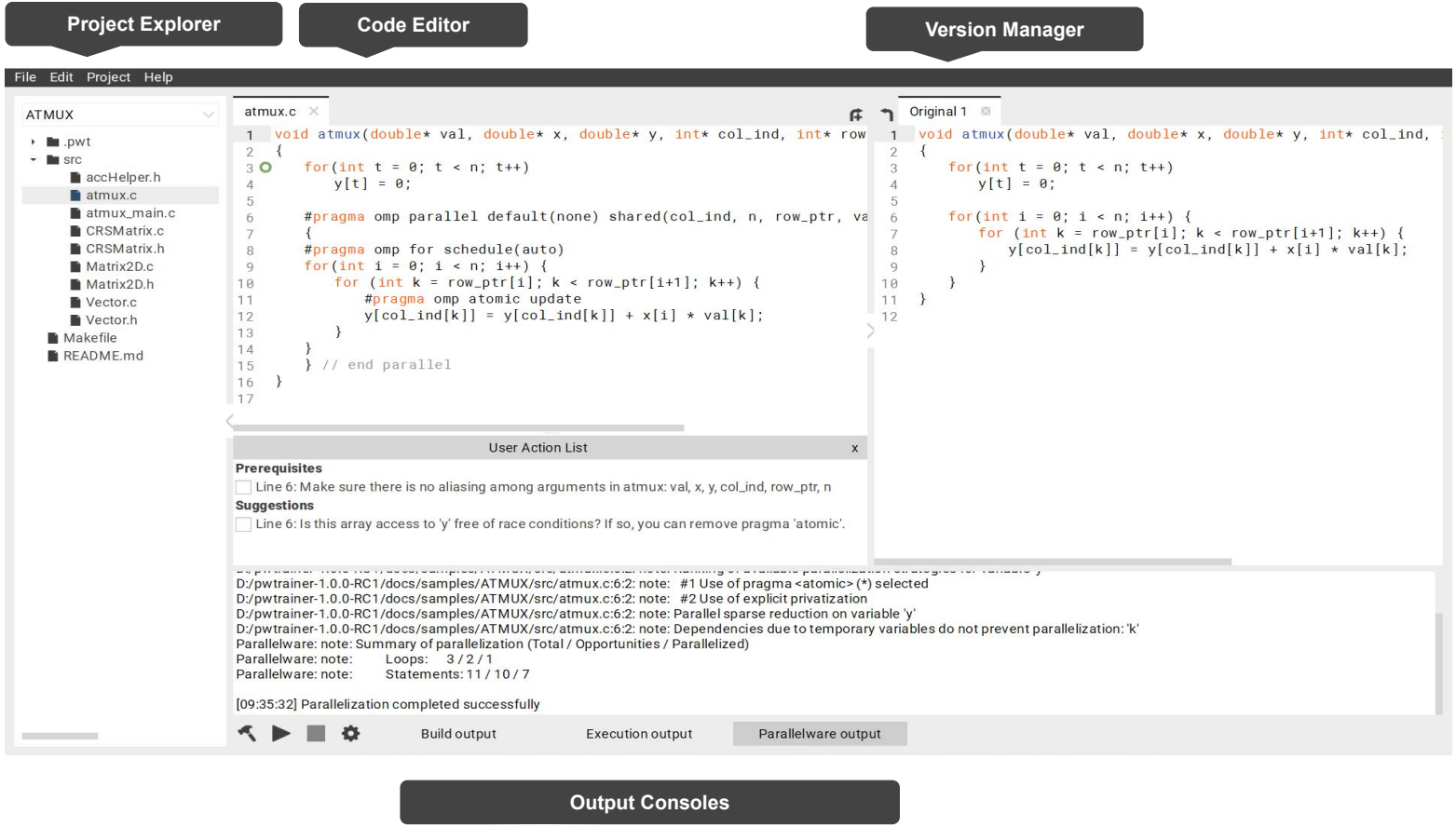
An interactive tool that acts as your mentor

> “Tell me, I will forget,
> Show me, I may remember,
> Involve me, I will understand.”

PARALLELWARE TRAINER

appentra
make code parallel

Parallelware Trainer



PARALLELWARE TRAINER

appentra
make code parallel

Parallelware Trainer

```
[09:29:21] Analysis completed: 1 opportunity found
[09:29:24] Parallelizing...
/home/User/examples/PI/src/pi.c:3:1: note: Analyzed function 'pi'
/home/User/examples/PI/src/pi.c:6:2: note: Parallel loop
/home/User/examples/PI/src/pi.c:6:2: note: Ranking of available parallelization strategies for variable 'sum'
/home/User/examples/PI/src/pi.c:6:2: note:   #1 Use of clause <reduction> (*) selected
/home/User/examples/PI/src/pi.c:6:2: note:   #2 Use of pragma <atomic>
/home/User/examples/PI/src/pi.c:6:2: note:   #3 Use of explicit privatization
/home/User/examples/PI/src/pi.c:6:2: note: Parallel reduction on variable 'sum' with associative, commutative operator '+'
/home/User/examples/PI/src/pi.c:6:2: note: Dependencies due to temporary variables do not prevent parallelization: 'x'
Parallelware: note: Summary of parallelization (Total / Opportunities / Parallelized)
Parallelware: note:          Loops:      1 / 1 / 1
Parallelware: note:          Statements: 7 / 5 / 5

[09:29:24] Parallelization completed successfully
[09:29:24] Analysis completed: 0 opportunities found
```

Build output | Execution output | Parallelware output

Analysis completed

PARALLELWARE TRAINER

Learn more

appentra
Software for guided parallelization

www.appentra.com

Stay in touch

Sign up for our newsletter:
www.appentra.com/blog/newsletter/

Email us at: info@appentra.com

Download/Purchase Parallelware Trainer:
www.appentra.com/products/parallelware-trainer/

Appentra appentra company/appentra/

Material for Trainers

- Decomposition of use cases into patterns
- Discussion of the training outcomes

Decomposition of PI into Patterns

Code file "pi.c"		Pattern			
Function	Line	Forall	Scalar reduction	Sparse reduction	Convergence loop
main()	29		sum		

Decomposition of LULESH into Patterns

Code file "luleshmk.c"		Pattern			
Function	Line	Forall	Scalar reduction	Sparse reduction	Convergence loop
CalcElemFBHourglassForce_workload()	60		sum		
CalcElemFBHourglassForce()	73	hgfx hgfy hgfz			
CalcFBHourglassForceForElems()	131			domain_m_fx domain_m_fy domain_m_fz	
ApplyMaterialPropertiesForElems_workload()	187		sum		
ApplyMaterialPropertiesForElems()	210	vnew			
	217	vnew			
CalcElemVelocityGradient_workload()	231		sum		
CalcKinematicsForElems()	258	domain_m_dxx domain_m_dzz	domain_m_dyy		
luleshmk()	292				iter (loop index)
main()	349	locDom_e locDom_m_dxx locDom_m_dyy	locDom_m_dzz vnew		
	358	locDom_m_nodelist			
	365	locDom_fx locDom_fy locDom_fz			
VerifyAndWriteFinalOutput()	85		MaxAbsDiff TotalAbsDiff MaxRelDiff		

HANDS-ON LAB: a walkthrough

- **Launch Parallelware Trainer:**
  - Login to the remote machine: *boada.ac.upc.edu*

    Connect using your account "*nct010XX*" (e.g. nct01026 - nct01055):

    $ ssh -YX <username>@boada.ac.upc.edu
  - Launch Trainer:

    $ /scratch/nas/1/marenaz/pwtrainer-0.5.3-x86_64-linux-ubuntu-14.04/pwtrainer &

- **Run the Pi calculation example:**
  - Copy the sample codes to your $HOME directory in boada:

    $ cp /scratch/nas/1/marenaz/samples.tgz $HOME
  - Open the PI project following instructions in the worksheet.

HANDS-ON LAB

Remote execution on “mt1.bsc.es” @BSC using SLURM

Add ssh key to avoid asking for password

- boada$ ssh-keygen// Generates new SSH key (press ENTER three times)
- boada$ ssh-copy-id <*username*>@mt1.bsc.es // Transfer new SSH key to mt1.bsc.es

Edit script “./samples/remote_on_mt1.config”

- Set “REMOTE_USER=<USERNAME>” // Your account “NCT010xx” at PATC
- Make sure variable “REMOTE_HOST=mt1.bsc.es”

Profile setup:Open the setup of the “Parallel” execution console

- Select “remote_run_on_mt1.sh” in “Custom execution script”
- Run the project (press F6 or click “Play” button)

DEMO with REMOTE execution!!

appentra
make code parallel